



Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencias de la Computación

Introducción a la Programación IIC1103

Prof. Ignacio Casas
icasas@ing.puc.cl

Tema 10 – Recursión Parte 1

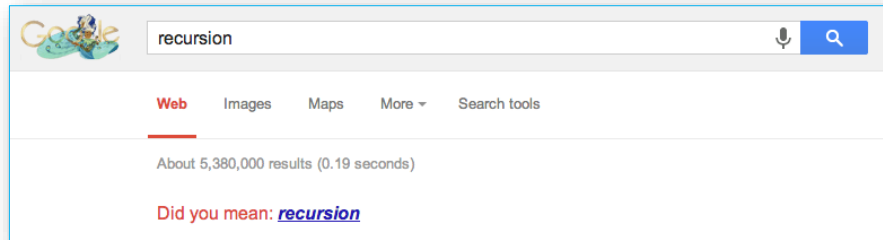
Colaboración de Mauricio Arriagada



Agenda

Recursión

¿Qué es la recursividad o recursión?



“Para entender la recursividad, antes tenemos que entender la recursión”

Una **oración** en español contiene un **sujeto** y un **predicado**. Un **predicado** contiene un **verbo**, un **objeto** y un **complemento**. Si consideramos 1.000 **sujetos** y 1.000 **verbos**, podemos construir 1.000.0000 de oraciones.

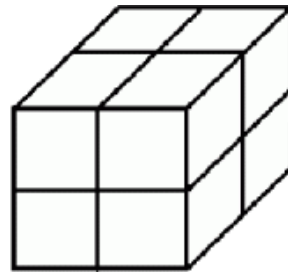
Cuidado con “irse por las ramas”: **recursión infinita**.

La recursividad según Escher



Dividir y simplificar para reinar ...

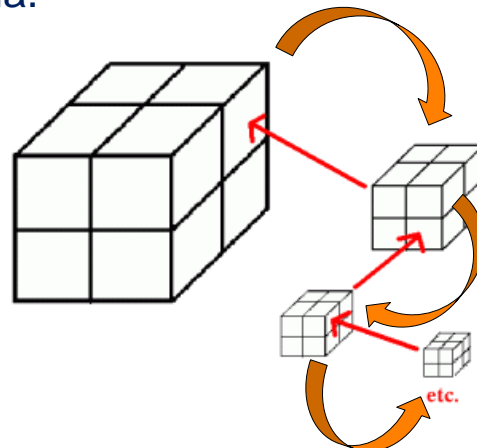
- **Recursión** es un método de resolución de problemas que se basa en dividir una situación en una “sucesión” de sub-problemas más sencillos, cada uno de los cuales se resuelve “invocando” a la misma solución.



Dividir y simplificar para Reinar ...

- La recursión funciona llamando al mismo **método/función** una y otra vez, **dentro de sí mismo**, cada vez resolviendo una parte más pequeña del problema.

Tiene que definirse un “límite” (**caso base**) para terminar las invocaciones.



Ejemplo “Cuenta Regresiva”

iteración vs recursión

```
""" Solución Iterativa """
def cuentatras(n):
    for i in range(n+1):
        j = n - i
        if j <= 0:
            print("Buuuummm!!!")
        else:
            print(j)
# programa ppal
cuentatras(10)
```

```
>>>
10
9
8
7
6
5
4
3
2
1
Buuuummm!!!
```

```
""" Solución Recursiva """
def cuentatras(n):
    if n <= 0:
        print("Buuuummm!!!")
    else:
        print(n)
        cuentatras(n-1)
# programa ppal
cuentatras(10)
```

Ejemplo “Cuenta Regresiva”

Caso base para
terminar la
recursión

```
""" Solución Recursiva """
def cuentatras(n):
    if n <= 0:
        print("Buuuummm!!!")
    else:
        print(n, " llamo a cuentatras con n = ", n-1)
        cuentatras(n-1)
# programa ppal
cuentatras(10)
```

```
>>>
10 llamo a cuentatras con n = 9
9 llamo a cuentatras con n = 8
8 llamo a cuentatras con n = 7
7 llamo a cuentatras con n = 6
6 llamo a cuentatras con n = 5
5 llamo a cuentatras con n = 4
4 llamo a cuentatras con n = 3
3 llamo a cuentatras con n = 2
2 llamo a cuentatras con n = 1
1 llamo a cuentatras con n = 0
Buuuummm!!!
```

Ejemplo “Cuenta Regresiva”

```
""" Solución Recursiva """
def cuentatras(n,s=""):
    if n <= 0:
        print(s+"Buuuummm!!!")
    else:
        print(s+str(n)+" llamo a cuentatras("+str(n-1)+")")
        s += " "
        a = cuentatras(n-1,s)
        print(s+"acabo de volver de "+str(n-1))
# programa ppal
cuentatras(10)
```

Agregamos un print después de la invocación a la función para observar el “retorno”.

Recordar que:

- (1) una función siempre “retorna” al lugar donde fue invocada, para continuar con la ejecución del código que la llamó.
- (2) La función **cuentatras()** retorna “None”.

```
>>>
10 llamo a cuentatras(9)
9 llamo a cuentatras(8)
8 llamo a cuentatras(7)
7 llamo a cuentatras(6)
6 llamo a cuentatras(5)
5 llamo a cuentatras(4)
4 llamo a cuentatras(3)
3 llamo a cuentatras(2)
2 llamo a cuentatras(1)
1 llamo a cuentatras(0)
Buuuummm!!!
acabo de volver de 0
acabo de volver de 1
acabo de volver de 2
acabo de volver de 3
acabo de volver de 4
acabo de volver de 5
acabo de volver de 6
acabo de volver de 7
acabo de volver de 8
acabo de volver de 9
```

Recursión “Infinita” (no es una buena idea)

```
""" Escher Infinito """
def Escher(p,s=""):
    print(s+" "+p+"? Es el hijo de Escher")
    Escher("Escher",s)
#
Escher("Escher","Quien es")
```

Python guarda un “stack” con todas las referencias a la función, para poder “regresar” una vez que llega al caso “base”.

Pero el tamaño en memoria de este stack es limitado y en algún momento se genera un error por “overflow” o “falta de memoria”.

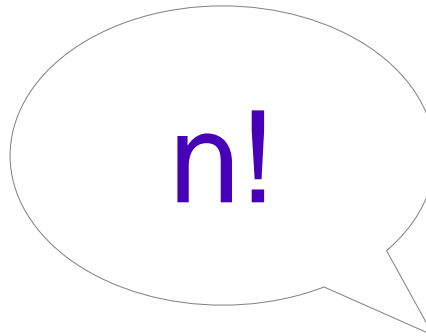
Nota:

Python no es capaz de detectar que en este programa no es necesario volver hacia atrás.

El porqué de la Recursión

- Se puede decir correctamente que la **recursión** es una **alternativa elegante** a la **iteración** (con **while** o **for**). (También podemos resolver el problema con una iteración.)
- Además de hacer “la vida más miserable a los alumnos”, la **ventaja** de dividir recursivamente un problema complejo en otros más pequeños es:
 - la solución (en muchos casos complejos) es más fácil de obtener por su “naturaleza recursiva”.
 - el código para la recursión es generalmente más corto que la solución iterativa.
- Pero la recursión en general **ocupa más memoria**. (Se debe mantener un registro (“stack”) de las llamadas sucesivas a la función, **para poder “volver atrás”**.) También puede demorar más en ejecutar.

Ejemplo: Factorial de n





n!

```

0! == 1
1! == 1
2! == 1*2
3! == 1*2*3
n! == 1*2*3*... *(n-1)*n
n! == n*(n-1)*(n-2)*...*2*1
3! == 3 * 2!
2! == 2 * 1!

```

Factorial(n)

Solución iterativa

```

def factorialIterativo(n):
    f = 1
    if (n == 0):
        return f
    i = n
    while (i>=1):
        f = f * i
        i = i-1
    return f

```

Para una solución recursiva:

¿Cómo podríamos dividir el problema en subproblemas?

¿Cómo podríamos dividir el problema en subproblemas?

$0! = 1 \rightarrow$ caso base (termina el cálculo)

$$n! = n * \underbrace{(n-1) * (n-2) * \dots * 2 * 1}_{(n-1)!}$$

Ejemplo: $5!$

$$5! = 5 * \underbrace{(5-1) * (5-2) * \dots * 2 * 1}_{(5-1)!}$$

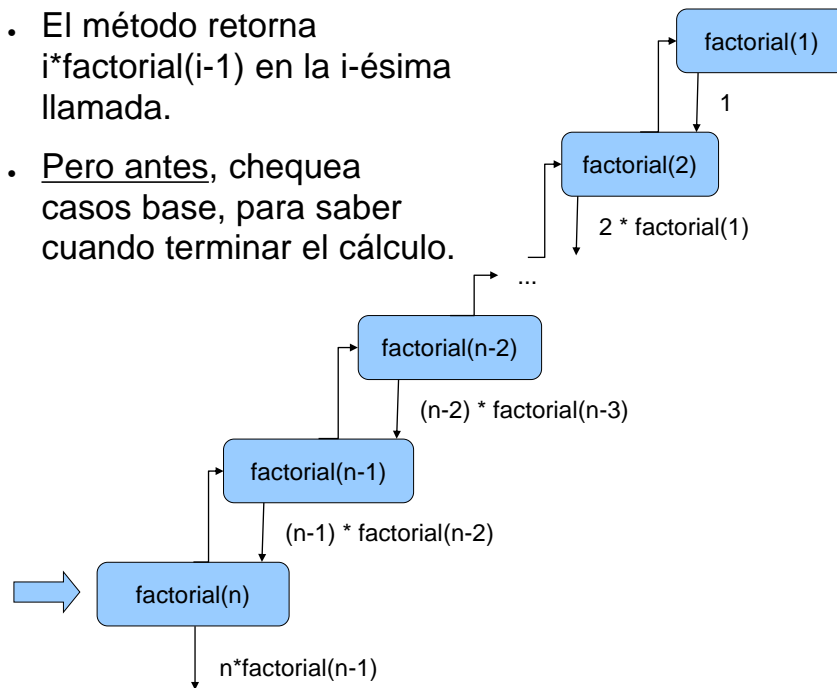
$$\begin{aligned}
 n! &= n * (\mathbf{n-1}) ! \\
 n! &= n * (n-1) * (\mathbf{n-2}) ! \\
 n! &= n * (n-1) * (n-2) * (\mathbf{n-3}) ! \\
 &\dots \\
 n! &= n * (n-1) * \dots * 2 * \underbrace{(1) !}
 \end{aligned}$$

1! → caso base (termina el cálculo)

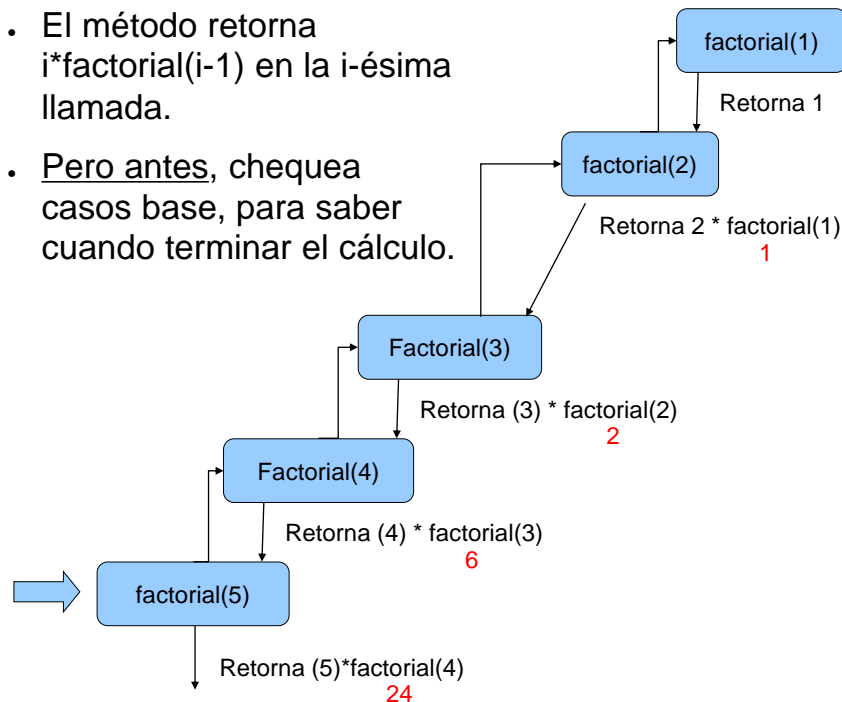
Solución recursiva

```
def factorialRecursivo(n):
    if (n == 0) or (n == 1):
        return 1
    return n * factorialRecursivo(n - 1)
```

- El método retorna $i * \text{factorial}(i-1)$ en la i -ésima llamada.
- Pero antes, chequea casos base, para saber cuando terminar el cálculo.



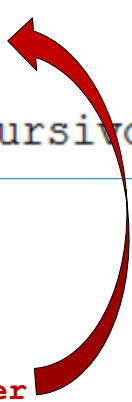
- El método retorna $i * \text{factorial}(i-1)$ en la i -ésima llamada.
- Pero antes, chequea casos base, para saber cuando terminar el cálculo.



Solución recursiva

```
def factorialRecursivo(n):  
    if (n == 0) or (n ==1):  
        return 1  
    return n * factorialRecursivo(n - 1)
```

CASOS BASE:
Indican cuándo detener
la recursión



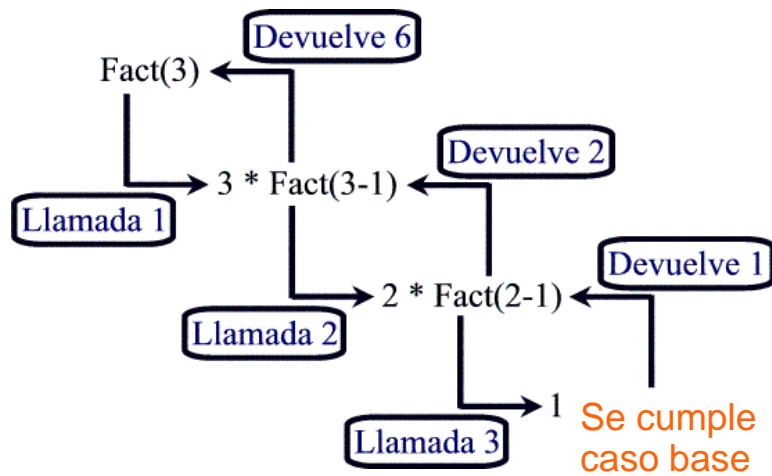
Solución recursiva

```
def factorialRecursivo(n):  
    if (n == 0) or (n ==1):  
        return 1  
    return n * factorialRecursivo(n - 1)
```

LLAMADA RECURSIVA,
cálculo y retorno



Ejemplo



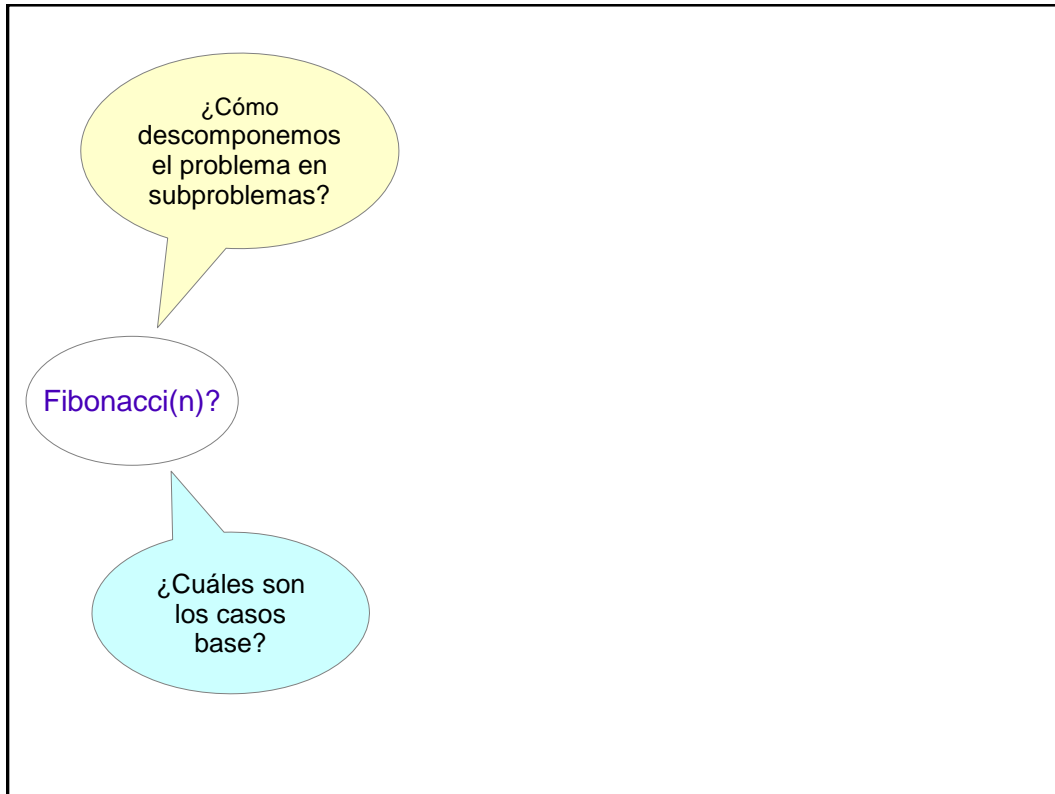
$F(3)$ $n=3$ return $3 * f(3-1)$
 $F(2)$ $n=2$ return $2 * f(2-1)$
 $F(1)$ $n=1$ return 1

Ejemplo: Serie de Fibonacci

Fibonacci(n)?



Escribir un método que determine
 el n -ésimo término de la sucesión
 de Fibonacci, en forma recursiva.



Fibonacci

- La Serie de Fibonacci es una sucesión infinita de números naturales, en la que el primer elemento es 1, el segundo es 1, y el i -ésimo corresponde a la suma de los dos anteriores. A cada elemento de esta sucesión se le llama número de Fibonacci:

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

- Se cumple que:

$$f[n] = f[n - 1] + f[n - 2]$$

¿Cómo
descomponemos
el problema en
subproblemas?

$$\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$$

Fibonacci(n)?

¿Cuáles son
los casos
base?

$$\cdot \text{fib}(0) = 1$$

$$\cdot \text{fib}(1) = 1$$

Solución

```
""" Fibonacci Recursivo """
def fibonacci(n):
    if (n == 0 or n == 1):
        return 1
    return fibonacci(n - 1) + fibonacci(n-2)
#
N = int(input("ingresa un entero >= 0: "))
F = fibonacci(N)
print("El número "+str(N)+" de Fibonacci es "+str(F))
```

Ejemplo: Suma de los dígitos de un número

Si tengo 2598
 $2+5+9+8=?$

Tarea: Escribir un método/función que sea capaz de sumar los dígitos de un número entero positivo, en forma recursiva.

Restricción (para complicar la vida de los estudiantes): **NO** usar strings ni listas.



Si tengo 2598
 $2+5+9+8=?$

¿Cómo obtenemos
los dígitos del
número?

¿Cómo descomponemos
la suma en
subproblemas?

¿Cuáles son los
casos base?

¿Cómo
obtenemos los
dígitos del
número?

Si tengo 2598
 $2+5+9+8=?$

Para obtener el último dígito del número:

Solución 1:

$\text{num} - (\text{num} / 10 * 10) \rightarrow 8$

Solución 2:

$\text{num} \% 10 \rightarrow 8$

Y el resto del número sería:

$\text{num} / 10 \rightarrow 259$

¿Cómo
descomponemos
la suma en
subproblemas?

Si tengo 2598
 $2+5+9+8=?$

- Podemos descomponer el número desde el último dígito hacia el primero.
- La suma de los dígitos es igual al último dígito más la suma de los dígitos del resto:

$(n \% 10) + \text{sumaDigitos}(n / 10)$

¿Cuáles son los casos base?

Si tengo 2598
 $2+5+9+8=?$

- Cuando el número sea sólo de un dígito, ya no habrá necesidad de descomponerlo, por lo que retornamos el mismo número.

$(n//10) == 0$

O bien:

$(n\%10) == n$

Solución recursiva

```
def sumaDigitos(num):  
    if (num // 10 == 0):  
        return num  
    return (num % 10) + sumaDigitos(num // 10)
```

Solución “tail recursion”

La “tail recursion” es una variante de la recursión. **NO deja operaciones pendientes** para el “regreso” desde el caso base. Algunos lenguajes pueden sacar provecho de esto para ser más eficientes... Python no lo tiene incorporado... aún.

Versión “tail recursion”:

```
def sumaDigitosTail(num, suma=0):
    if (num // 10 == 0):
        return suma + num
    return sumaDigitosTail(num // 10, suma+(num%10))
print(sumaDigitosTail(249017))
```

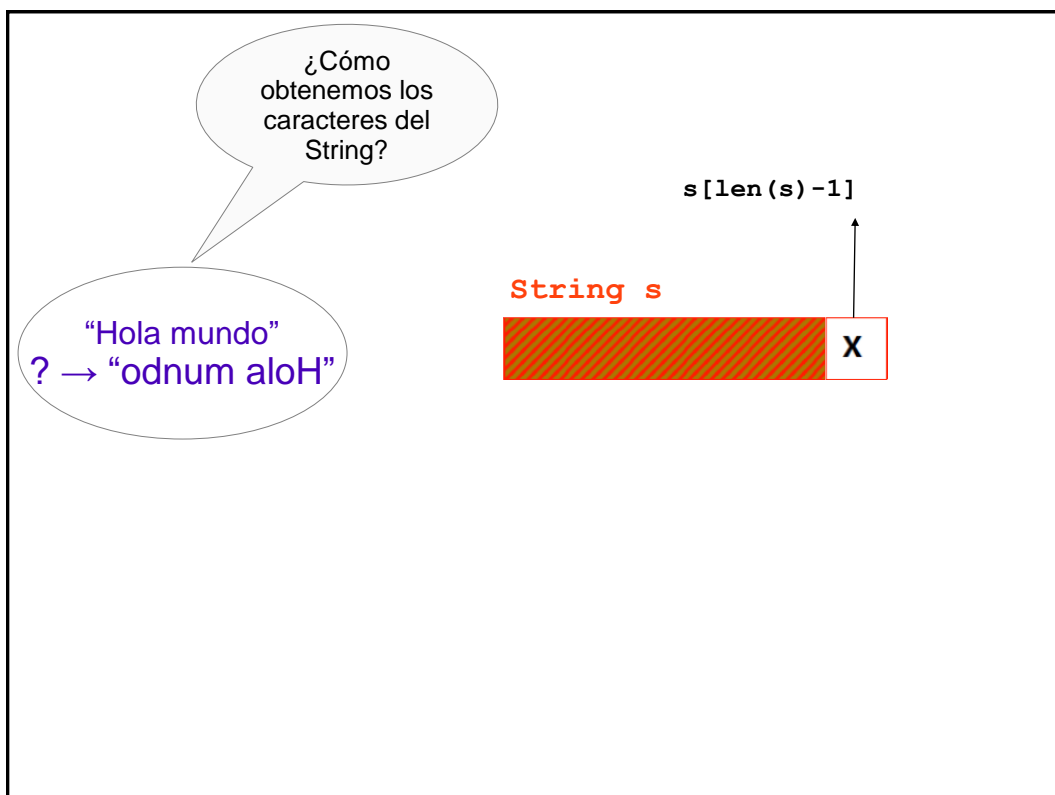
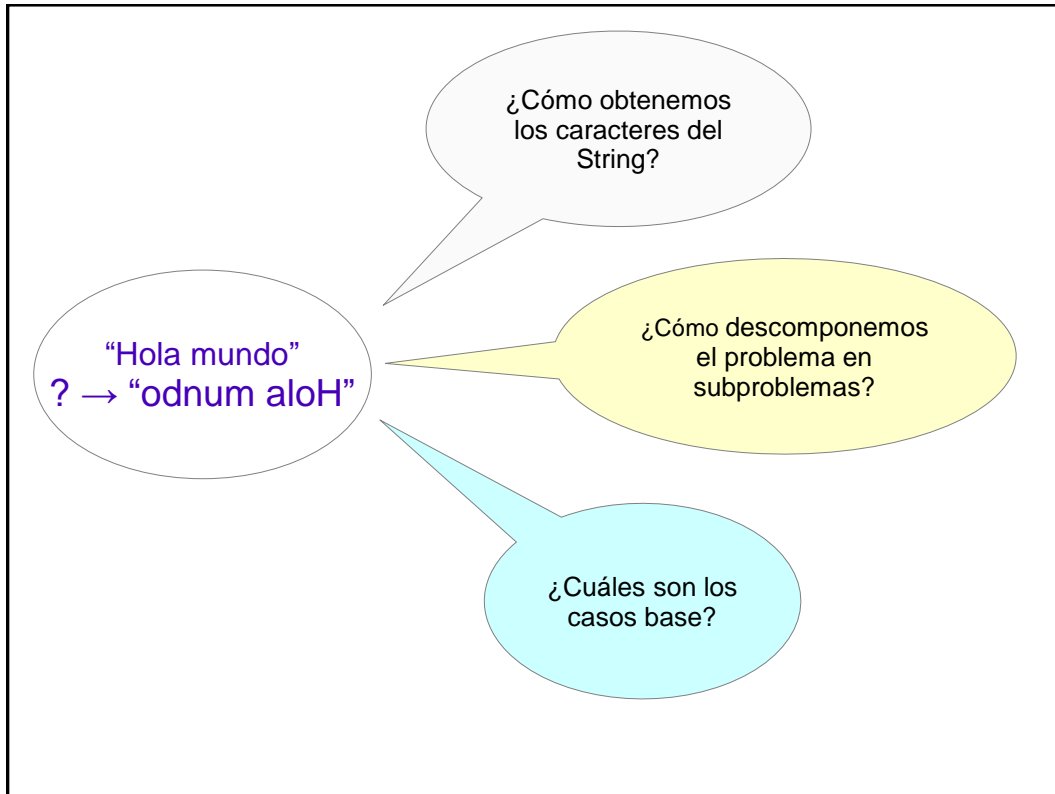
Ejemplo: Invertir un string

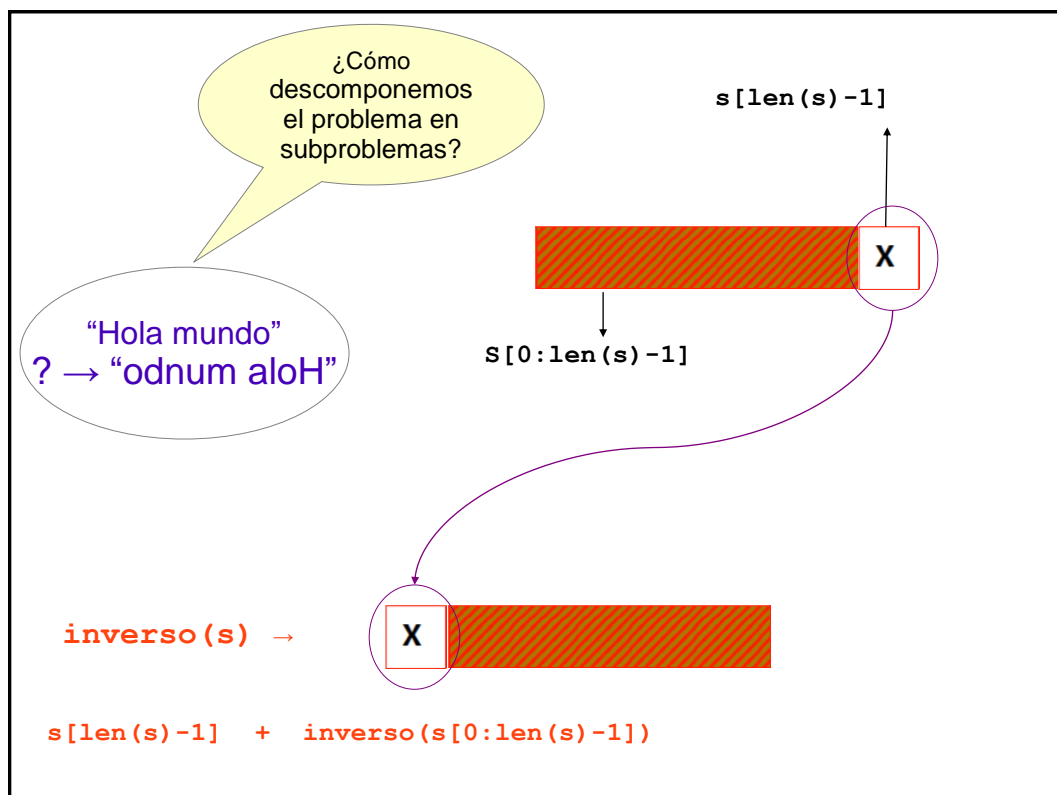
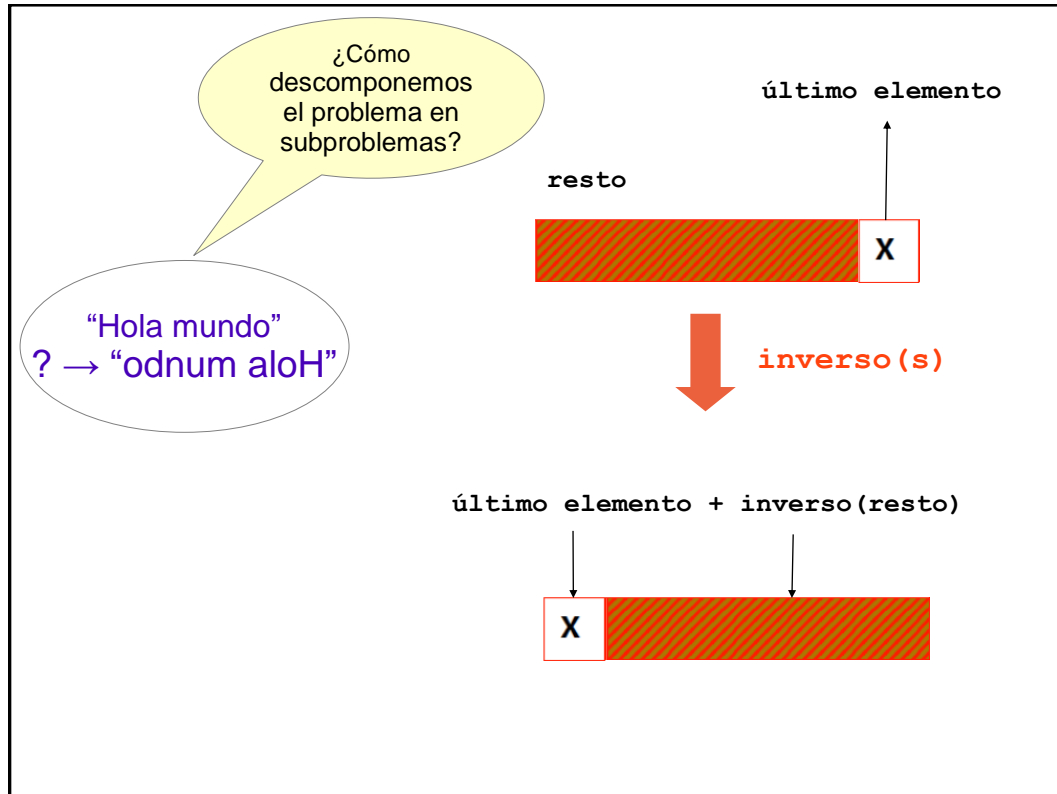
“Hola mundo”
? → “odnum aloH”

Se pide escribir un método que reciba como parámetro un string, y retorne su reverso (la misma palabra o frase, pero invertida).

Hacerlo de manera recursiva (sin usar métodos pre-definidos de strings ni listas).







¿Cuáles son los casos base?

"Hola mundo"
? → "odnum aloH"

• Si el string es **vacío** o de **tamaño 1**:

`if (s == "") → inverso = ""`

`if (len(s) == 1) → inverso = s`

Solución

```
def invertirString(s):
    if (s == "" or len(s) == 1):
        return s
    return s[(len(s)-1)] + invertirString(s[0:len(s)-1])
```

llega hasta len(s)-2

Versión "tail recursion":

```
def InvStringTail(s, result=""):
    if s == "":
        return result
    return InvStringTail(s[0:len(s)-1], result+s[len(s)-1])
```

Ejemplo: Palíndromo

¿“oso” es palíndromo?



- Se pide escribir un método que reciba como parámetro un string, y determine si es palíndromo o no.
- Hágalo de manera recursiva.

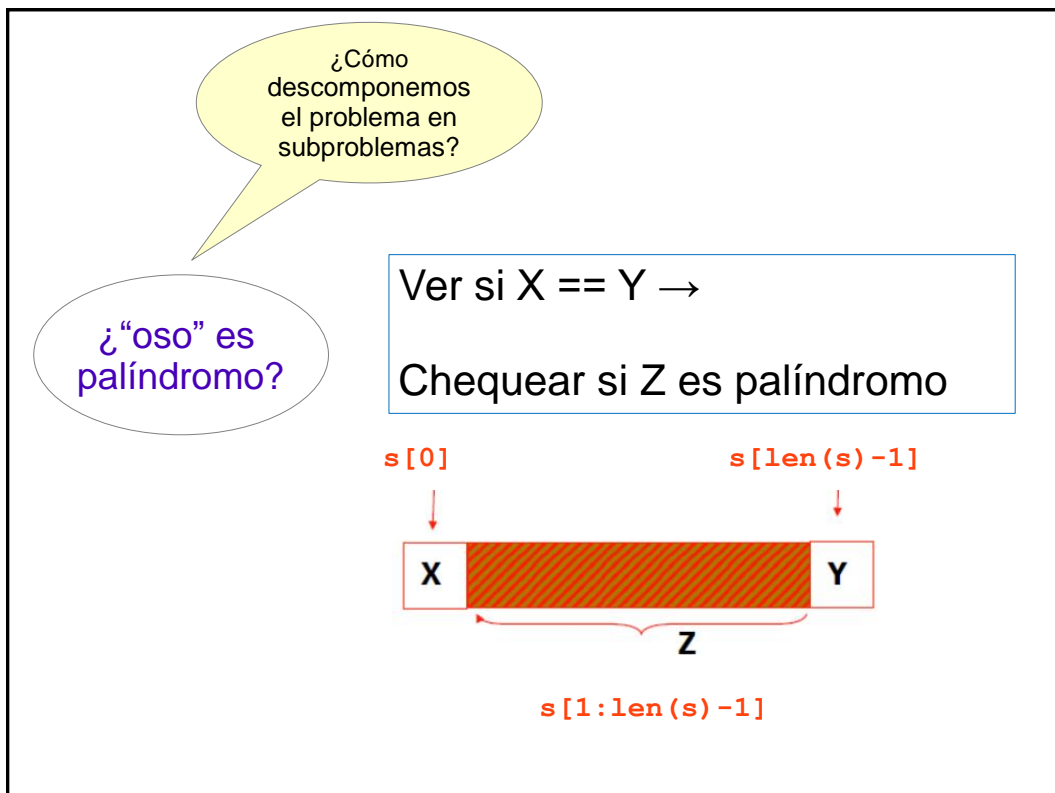
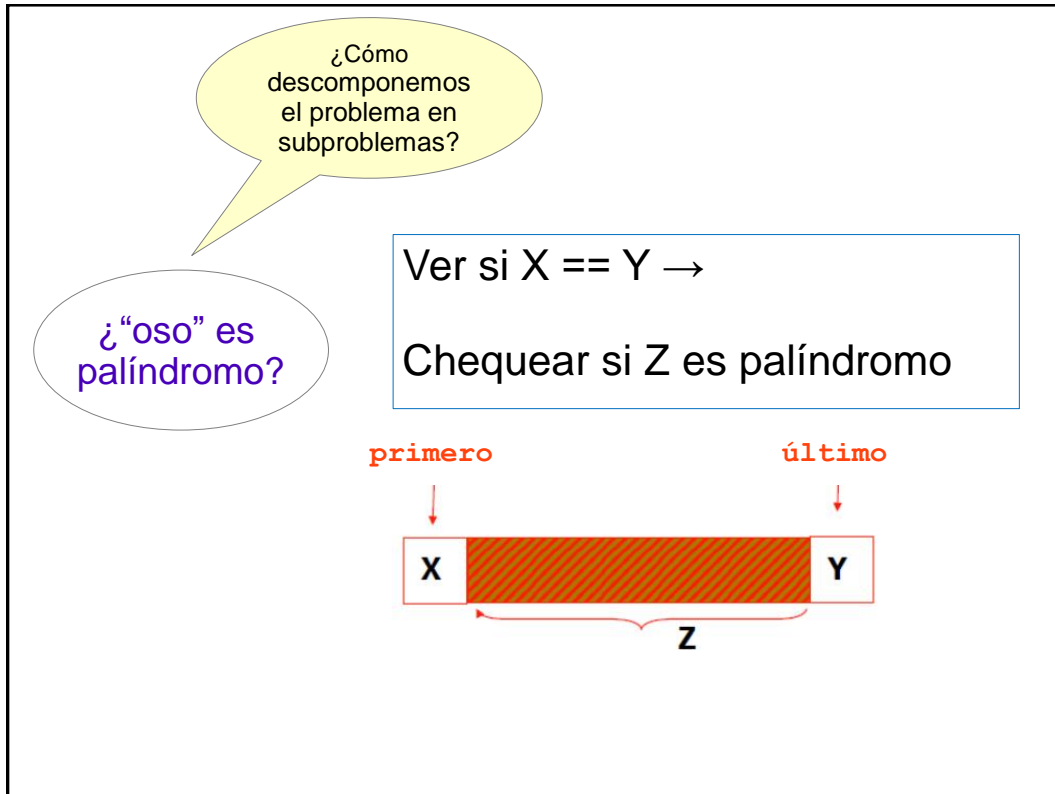
¿“oso” es palíndromo?

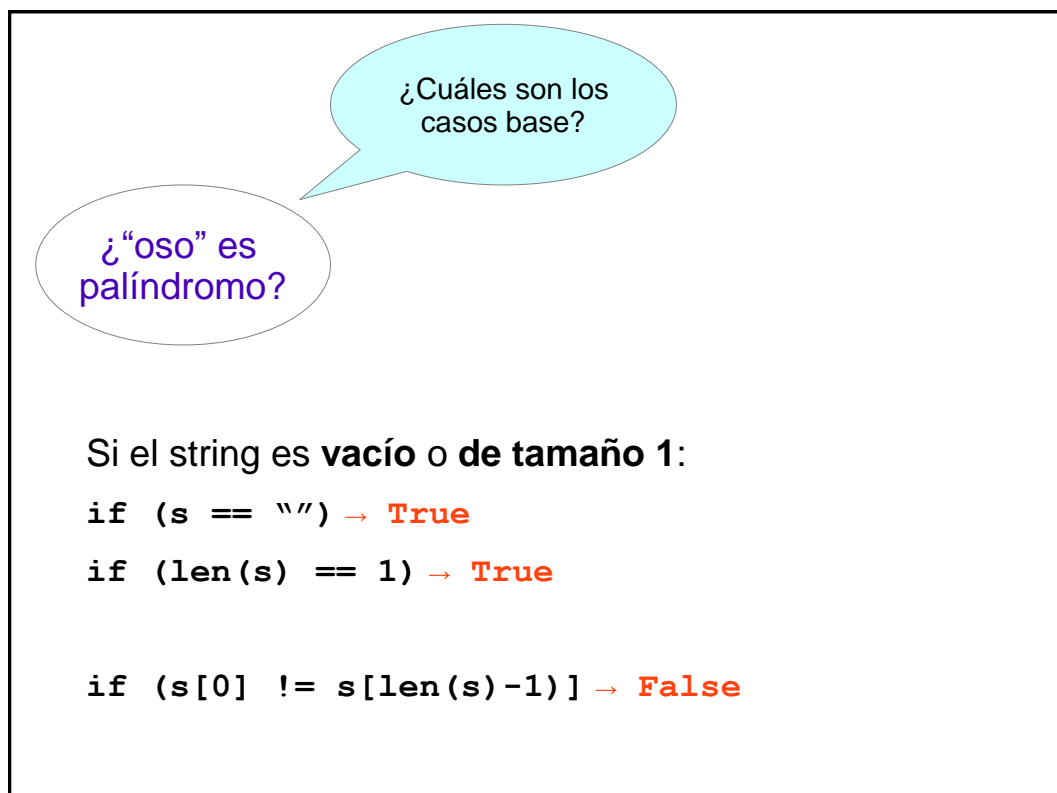
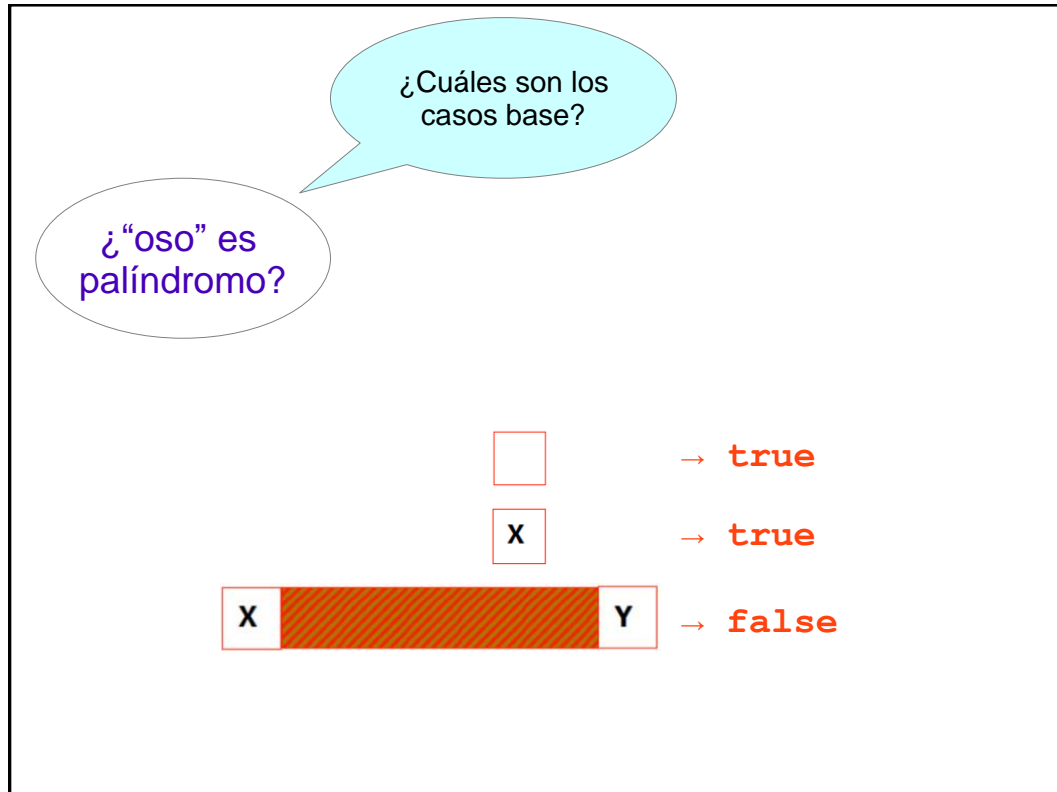
¿Cómo obtenemos los caracteres del String?



¿Cómo descomponemos el problema en subproblemas?

¿Cuáles son los casos base?





Solución

```
def esPalindromo(s):
    s = s.replace(" ", "")
    if (len(s) <= 1):
        return True
    if (s[0] != s[len(s)-1]):
        return False
    return esPalindromo(s[1:len(s)-1])
```

elimina todos los blancos

Nota 1: Este algoritmo es “tail recursion” pues no deja operaciones pendientes. (Pero igual Python “regresa” de todas las invocaciones). 😊 ☹️

Nota 2: las invocaciones recursivas se detienen cuando se encuentra el primer **False**... y luego se hace el “regreso”. 😊

Mini-Tarea 14: Un Desafío

Desafío

Describe de forma clara y precisa qué es lo que hace la siguiente función.

Sugerencia:
Escribe el programa y pruébalo para distintas listas de números enteros. Agrégale varios print() y sigue su ejecución paso a paso.



Recursión Misteriosa

```
""" recursión misteriosa """
def misterio(lista,i,j,x):
    if (lista[(i+j)//2] == x):
        return True
    if (i==j):
        return False
    else:
        return (misterio(lista,i,(i+j)//2,x) or
                misterio(lista,(i+j)//2+1,j,x))
#
lista = [31,6,12,762,45,34,87,56,1,86]
x = 9
print(x,misterio(lista,0,(len(lista)-1),x))
x = 87
print(x,misterio(lista,0,(len(lista)-1),x))
|
```

Describe de forma clara y precisa
qué es lo que hace esta función.



Variante 1 del Misterio

```
""" recursión misteriosa """
def misterio(lista,i,j,x):
    if (lista[(i+j)//2] == x):
        return True
    if (i>=j):
        return False
    else:
        return (misterio(lista,i,(i+j)//2-1,x) or
                misterio(lista,(i+j)//2+1,j,x))
#
lista = [31,6,12,762,45,34,87,56,1,86]
x = 9
print(x,misterio(lista,0,(len(lista)-1),x))
x = 87
print(x,misterio(lista,0,(len(lista)-1),x))
```



Variante 2 del Misterio

```
""" variante de la recursión misteriosa """
def misterioBin(lista,i,j,x):
    if (lista[(i+j)//2] == x):
        return True
    if (i>=j):
        return False
    elif (x < lista[(i+j)//2]):
        return misterioBin(lista,i, (i+j)//2-1,x)
    else:
        return misterioBin(lista, (i+j)//2+1,j,x)
#
lista = [1,6,12,31,34,45,56,86,87,762]
x = 9
print(x,misterioBin(lista,0,(len(lista)-1),x))
x = 87
print(x,misterioBin(lista,0,(len(lista)-1),x))
```



Todos los ejemplos anteriores fueron resueltos con:

RECURSION "SIMPLE":

seguimos un único camino posible para encontrar la solución, sub-dividiendo en problemas más simples. No exploramos opciones.

→ Primero identificamos los "**casos base**" (para "parar" la recursión) y luego realizamos la llamada recursiva.

→ Las llamadas recursivas **terminan en el caso base** y luego se regresa de cada invocación, entregando el resultado en el "último regreso" del método/función.

La "**TAIL RECURSION**" es una mini-variante que no deja operaciones pendientes para el retorno.



Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencias de la Computación

Introducción a la Programación IIC1103

Prof. Ignacio Casas
icasas@ing.puc.cl

Tema 10 – Recursión Parte 1

Colaboración de Mauricio Arriagada